ECE-6483 Real Time Embedded Systems
Homework 3

New York University, Spring 2025

Due on 03/11, 11:59 PM

Name: Raman Kumar Jha
NYU ID: N13866145

In class, we talked about memory mapped I/O. In memory mapped I/O, peripherals are accessed by writing to certain addresses in memory, using the same instructions as for other memory access. As embedded systems engineers, you may not always have the luxury of accessing GPIO peripherals using well-documented, mature libraries for popular microcontrollers. You may be asked to design a new embedded system, to write firmware libraries for accessing GPIO on a new embedded system, or to use poorly documented in-house libraries for accessing GPIO. The aim of this homework, therefore, is to help you gain a deeper understanding of how memory mapped I/O works so that you will be capable not only of using GPIO libraries, but also writing and documenting them.

1. Please refer to the Cortex M-4 Technical Reference Manual and the STM32F4 Discovery Reference Manual (attached to this HW) to answer this question.

(a) Section 3.4 in the Cortex M-4 Technical Reference Manual defines the memory map for this processor. What range of memory addresses is reserved for use by peripherals?

Solution: The peripheral memory address range for Cortex M-4 is from $0x40000000$ to $0x5FFFFFFF$.

(b) Section 2.3 of the STM32F4 Discovery Reference Manual further defines the memory used by each individual peripheral. The GPIO peripherals are listed by port (GPIOA, GPIOB, GPIOC, GPIOD, etc.). What is the address range used by the GPIOD port?

Solution: The address range used by GPIOD port is $0x40020C00 - 0x40020FFF$.

(c) On the STM32F4 Discovery board, each GPIO pin gets some dedicated configuration and data registers. These are described in Section 8, and es-

pecially Section 8.4, of the STM32F4 Discovery Technical Reference Manual. Please answer these questions for the following registers: MODER, OTYPER, OSPEEDR, PUPDR, IDR, ODR, and BSRR. For the BSRR register, you should further subdivide into BSRRL (BSRR low, used for bit set) and BSRRH (BSRR high, used for bit reset).

- The functionality of the register; what is it used for (in one sentence)?
- How much memory is used by this register for the entire port (typically 32 bits - 4 bytes - or 16 bits - 2 bytes)? How much memory is used for each individual pin (typically 2 bits or 1 bit)?
- What is the address offset of this register? The address offset, together with the port address, tells us exactly where in memory the register is located. For example, if the address offset of a register is 0x02 and the port uses the memory range 0x40022800 - 0x40022BFF, then the register is at address 0x40022800 + 0x02 = 0x40022802.

Solution:

MODER

- **Functionality:** Sets mode of pins (input/output/alternate function/analog).
- **Size:** 32 bits (4 bytes) total; 2 bits per pin.
- **Address offset:** $0x00$

OTYPER

- **Functionality:** Selects output type (push-pull or open-drain).
- **Size:** 16 bits total (2 bytes), 1 bit per pin.
- **Address offset:** $0x04$

OSPEEDR

- **Functionality:** Configures output speed.
- **Size:** 32 bits total (4 bytes), 2 bits per pin.
- **Address offset:** $0x08$

PUPDR

- **Functionality:** Configures pull-up/pull-down resistors.
- **Size:** 32 bits total (4 bytes), 2 bits per pin.
- **Address offset:** $0x0C$

IDR

- **Functionality:** Input data register; reads input state of pins.
- **Size:** 16 bits total (2 bytes), 1 bit per pin.
- **Address offset:** $0x10$

ODR

- **Functionality:** Output data register; sets output level.
- **Size:** 16 bits total (2 bytes), 1 bit per pin.
- **Address offset:** $0x14$

BSRRL

- **Functionality:** Bit set/reset register low; used to set pins.
  Lower half of BSRR register; writing '1' sets corresponding pin.
- **Size:** 16 bits total (2 bytes), 1 bit per pin.
- **Address offset:** $0x18$

BSRRH

- **Functionality:** Bit set/reset register high; resets pins when set to '1'.
- **Size:** 16 bits total (2 bytes), 1 bit per pin.
- **Address offset:** $0x1A$

2. This is a continuation of the previous question. Now that we know exactly where in memory each register is located, we can write C code to access these memory locations.
When you define a structure in C, the memory for elements of the structure will be laid out in the same order in which you defined them. Given that the registers are located sequentially in memory, we can use a structure in C to map registers to human readable names.

(a) We will use a GPIO_TypeDef struct to map memory for GPIO registers to human readable names. Here's a template, showing the struct definition and also how we define GPIOD using the struct:

```
typedef struct
{
} GPIO_TypeDef;

#define GPIOD_BASE (0xAAAAAAAA) // insert correct memory address here
#define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
```

Inside the typedef, we will add a line for each register that defines its total size for the entire port (32 bits - uint32_t - or 16 bits - uint16_t), its name, and a comment giving its functionality. In order to make sure the struct element is located at the correct memory offset, we need to make sure to define them in order. We will also define each element as volatile, to signal to the compiler that they may be modified outside of our code (e.g., by peripherals) and shouldn't be optimized away.

For example, the MODER register occupies 32 bits, and is located at an address offset of $0x00$. So we have to define it as the first element of the structure, and as a uint32_t. After adding MODER, our typedef would look like this:

```
typedef struct
{
    volatile uint32_t MODER; // set port mode, e.g. input, output
} GPIO_TypeDef;

#define GPIOD_BASE (0xAAAAAAAA) // insert correct memory address here
#define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
```

and we would be able to access the GPIOD MODER register as `GPIOD->MODER`.

(b) Please fill out the following C template. Add an element to the structure for each of: MODER, OTYPER, OSPEEDR, PUPDR, IDR, ODR, BSRRL, and BSRRH. Also, set the correct value for the GPIOD_BASE address based on your answer to question 1(b).
Note that the elements in a struct must be defined in the correct order and with the correct sizes to ensure they are placed at the intended memory locations based on the offsets specified in the datasheet.

For example, given the definition below:

- The struct begins at the memory location defined by `GPIOD_BASE`, which is `0x40020C00`.
- The first element in the struct is at `0x40020C00` with an offset of `0x00`. Writing to `GPIOD->MODER` corresponds to writing to memory location `0x40020C00`.
- The second element, `OTYPER`, is located at `0x40020C00` plus the size of `MODER`. Since `MODER` is 32 bits (4 bytes), `OTYPER` is located at `0x40020C04`.
- The third element, `PUPDR`, is located at `0x40020C00` plus the combined size of `MODER` and `OTYPER`. Since both are 32 bits (4 bytes each), `PUPDR` is at `0x40020C08`.
- Using a similar process, we determine the memory locations of all subsequent struct elements. Note that some elements are defined as 16-bit values, which affects their memory alignment.

If you refer to `stm32f4xx.h`, you will find a similar struct definition. This demonstrates how a basic GPIO peripheral driver can be written using just the datasheet for reference.

Solution:

The completed struct definition is:

```
typedef struct
{
    volatile uint32_t MODER;   // GPIO port mode register (offset: 0x00)
    volatile uint32_t OTYPER;  // GPIO port output type register (offset: 0x04)
    volatile uint32_t OSPEEDR; // GPIO port output speed register (offset: 0x08)
    volatile uint32_t PUPDR;   // GPIO port pull-up/pull-down register (offset: 0x0C)
    volatile uint32_t IDR;     // GPIO port input data register (offset: 0x10)
    volatile uint32_t ODR;     // GPIO port output data register (offset: 0x14)
    volatile uint16_t BSRRL;   // GPIO bit set/reset low register (offset: 0x18)
    volatile uint16_t BSRRH;   // GPIO bit set/reset high register (offset: 0x1A)
} GPIO_TypeDef;

#define GPIOD_BASE (0x40020C00)
#define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
```

This ensures each register maps correctly according to their offsets.

(c) Which element(s) of the struct do you think you might declare as a `volatile const` instead of just a `volatile`? Why?

Solution:

The IDR (input data register) could be declared as `volatile const` because it's read-only from software perspective. Its value is determined externally by hardware inputs and should not be modified by software.

(d) Do we need to use malloc() in C to allocate memory for GPIOD? Why or why not?

Solution:

No, We do not use `malloc()` because memory mapped registers have fixed physical addresses predefined by hardware specifications. We simply point our struct pointer directly at these addresses.

(e) Please answer these questions for registers MODER, OTYPER, OSPEEDR, PUPDR, ODR, BSRRL, and BSRRH:

- For all registers except IDR, write a line of C code that assigns '1' to this register for pin 3 of GPIOD without affecting other pins.

- For all registers except IDR, write a line of C code that assigns '1' to this register for all pins.
- What does setting a value of '1' do on this register?

Solution:

Setting Values for Registers To set values only for pin 3:

```
GPIOD->MODER |= (1 << (3 * 2));     // Set mode for pin 3
GPIOD->OTYPER |= (1 << 3);          // Set output type for pin 3
GPIOD->OSPEEDR |= (1 << (3 * 2));   // Set speed for pin 3
GPIOD->PUPDR |= (1 << (3 * 2));     // Set pull-up/pull-down for pin 3
GPIOD->ODR |= (1 << 3);             // Set output data for pin 3
GPIOD->BSRR |= (1 << 3);            // Set bit for pin 3
```

To set values for all pins:

```
GPIOD->MODER = 0xFFFFFFFF;    // Set mode for all pins
GPIOD->OTYPER = 0xFFFF;       // Set output type for all pins
GPIOD->OSPEEDR = 0xFFFFFFFF; // Set speed for all pins
GPIOD->PUPDR = 0xFFFFFFFF;    // Set pull-up/pull-down for all pins
GPIOD->ODR = 0xFFFF;          // Set output data for all pins
GPIOD->BSRR = 0xFFFF;         // Set bit for all pins
```

Setting a value of '1' in these registers enables specific configurations, such as setting a pin as an output, enabling pull-up resistors, or writing high values to outputs.